(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 014 265 A1

(12) **EUROPEAN PATENT APPLICATION**

(43) Date of publication:
28.06.2000 Bulletin 2000/26

(51) Int Cl.[7]: **G06F 11/00**

(21) Application number: **99309818.5**

(22) Date of filing: **07.12.1999**

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: **15.12.1998 US 211967**

(71) Applicant: **LUCENT TECHNOLOGIES INC.**
Murray Hill, New Jersey 07974-0636 (US)

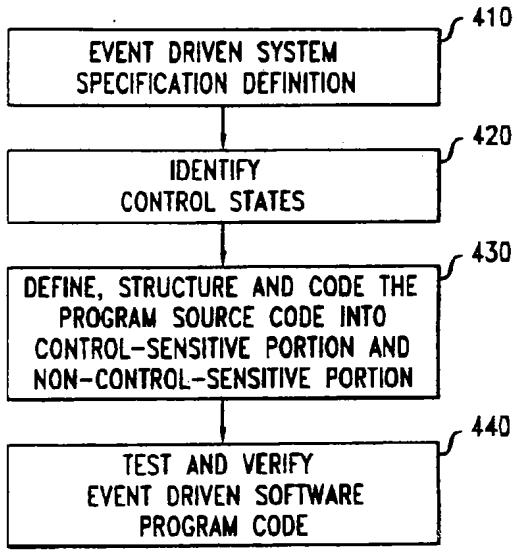
(72) Inventors:
- **Holzmann, Gerard Joan**
  Murray Hill, New Jersey 07974 (US)
- **Lane Thompson, Kenneth**
  Watchung, New Jersey 07060 (US)
- **Winterbottom, Philip Steven**
  Watchung, New Jersey 07060 (US)

(74) Representative:
Watts, Christopher Malcolm Kelway, Dr. et al
Lucent Technologies (UK) Ltd,
5 Mornington Road
Woodford Green Essex, IG8 0TU (GB)

(54) Method and apparatus for testing event driven software

(57) A technique for testing event driven software. In accordance with the technique, the source code of the event driven software is directly converted to a automaton based model useful in verifying that the program code complies with the desired properties defined by the user. More particularly, the event driven system program code is translated into a target language for a particular model checker. Such a translation results in a model which contains statements directed at whether execution of the program code will affect the behavior of the event driven system. Thus, this model extraction process can be used as input to a logic model checker for determining whether event driven system complies with the desired correctness properties specified by the user. Advantageously, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the subject event driven system program code without the need for user intervention.

FIG. 4



EP 1 014 265 A1

Description

Field of the Invention

[0001] The present invention relates to testing computer software programs and, more particularly, to a technique for testing event driven application programs.

Background of the Invention

[0002] In well-known event driven software systems (also referred to in the art as "reactive systems"), certain functionality occurs when an event is generated and communicated to an executing process performing particular actions based upon the occurrence of the particular event. More particularly, event driven software is used to define the behavior of a system, e.g., a computer, in response to events that are generated by other systems, machines or peripheral devices connected thereto. Well-known examples of event driven systems include: telecommunications applications, call-processing software, data communications software, device drivers, and computer interface applications. As will be appreciated, these types of event driven systems are state oriented. That is, in each state of a system, one specific event from a predefined set of events is expected to occur, and the system is designed to respond to the event in a well-defined manner.

[0003] Briefly, as is well-known, a state machine converts a time series of event stimuli or input data through some function into a time series of responses or output data. For example, in an event driven system, at each point in time, each machine is in a particular so-called "control state" which defines the events that are expected to be generated by other machines, and the type of responses that are to be generated upon the arrival of each such event. Typically, the response includes the execution of a finite piece of program code, the generation of new events, and the identification of a new control state for the particular machine responding to the incoming events. Thus, the state machine model of the event driven system plays a critical role in the development of the application program code by the programmer for the system.

[0004] Typically event driven application software is written in higher level general-purpose programming languages such as the well-known "C" or "C++" programming languages, or a special-purpose language such as the well-known ITU Systems Description Language ("SDL"). In developing the actual program code for such systems it is common to describe the behavior of the relevant machines in terms of state machines. Using a higher level programming language, e.g., C, software programmers will write event driven software code for the particular application. Typically, the software program development begins by reviewing a so-called system specification which provides a logical description of the desired operations for the application. Working from the specification, one or more programmers will write the necessary code to implement the operations.

[0005] This conventional programming process results in an application program which contains no particular visual context amongst and between the various control states of the event driven system. That is, the program code can be viewed as one large "flat" file with a series of program instructions. Further, as the original specification changes during program development, e.g., additional functionality is added to the system, the requisite program code and additional control states are typically added to the source program in a somewhat arbitrary order. In order for a programmer to understand the logical flow from control state-to-control state directly from the program source code it is necessary to manually trace the execution thread through the source, e.g., tracing branch instructions or "goto" statements. As will appreciated, the tracing of such logical flow, particularly in very large source code programs, is a tedious and time consuming task. This also leads to certain complexity in the testing of the software for so-called "bugs" and to determine whether the application program implements the desired features.

[0006] More particularly, a critical state in the development of the application software code for such systems is the testing of the code to validate the properties of the software and to insure that the behavior of the system complies with the design objectives. There are a number of well-known methods for testing event driven software. For example, Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems,* John Wiley and Sons, 1984, describes a well-known testing arrangement wherein a set of test goals is defined from which a limited set of test sequences is deduced. Typically, the set of test sequences is defined through a manual process by the testing engineer through examination of the system design. The set of test sequences is applied to the event driven software under test and either a "pass" or "fail" is recorded as the testing result. This testing methodology is typically applied to general software testing and is well-known in the art to be prone to human error and incomplete. In particular, the number of possible executions of the event driven software under test is incredibly large. In contrast, the number of tests that can be performed on the software with real-time executions is limited by certain practical constraints such as the minimum time required to perform a single test from the total number of tests. As such, it is typical that no more than a few hundred tests can be selected and applied in accordance with this methodology. This results in a very low confidence level in the testing coverage.

[0007] A second event driven software method known in the art is described, e.g., by D. Lee et al, Principles and Methods of Testing Finite State Machines - A Survey, *Proceedings of the IEEE,* Vol. 84, pp. 1090-1126, 1996. In accordance with this testing method, an abstract state machine model for the event drive software program is constructed. The state machine model typically captures only a high-level description of the program code thereby leaving out most of the program details. Thereafter, well-known test sequence generation algorithms (see, e.g., Lee, *supra.*) can be applied to the state machine model from which a relatively small set of tests is derived and applied to the software under test. As with the above-described testing method, this technique relies heavily on the manual construction of the abstract model of the event driven software. Therefore, this technique is sensitive to certain errors caused by mismatches between the formal relation between the software under test and the abstract model to be used for deriving the applicable tests. In addition, changes to the event driven software require the development of new tests which proves cumbersome and time consuming.

[0008] In a third commonly used testing method, see, e.g., C. H. West, Protocol Validation by Random State Exploration, *Proc. 6th IFIPWG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification,* North-Holland Publ., pp. 233-242, 1986, a random exploration of the full behavior of the event driven software is undertaken in order to establish the software's compliance with the desired properties. The amount of detail in the full software description generally precludes any attempt to perform an exhaustive exploration of the software for testing purposes. Therefore, well-known statistical algorithms are used to extract and analyze random samples of the software's behavior. However, as with the other above-described testing arrangements, one recognized drawback in this testing approach is that the amount of extraneous detail in the full software description prevents an efficient exhaustive exploration of the full behavior of the software for compliance with high-level properties.

[0009] Therefore, a need exists for a technique that mitigates the above-described problems in the art and improves the programming development and testing of event driven software.

Summary of the Invention

[0010] The present invention provides a technique for testing event driven software directed to the efficient testing of event driven software and making it possible to track the validity of future changes to the program code, as part of the normal maintenance, extension and revision process to the source code. More particularly, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the co-pending, commonly assigned patent application, entitled "Method and Apparatus For Developing Event Driven Software", Application Serial No. , filed on event date herewith. That is, in accordance with the preferred embodiment, the source code program is defined and structured into control-sensitive and non-control-sensitive format. In accordance with the present invention, the event driven source code program is parsed to extract the control states defined in the source program, and to convert the source code into an intermediate state machine format. Thereafter, in accordance with the invention, the intermediate state machine format is converted into a automata-based format for modeling checking purposes.

[0011] In particular, in accordance with the preferred embodiment, the intermediate state machine format is converted into an annotated automaton model. Illustratively, the annotated automaton model is a verification program written in the input language of a specific model checker. Importantly, in accordance with the invention, each statement from the source code program is mapped onto an abstraction within the automaton model. The mapping is achieved using a translation map which is defined at the initial stage of the verification process. The map need not be revised thereafter unless new types of instructions are introduced, e.g., after revisions are made to repair programming faults, into the source program. The mapping, *inter alia,* dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the source code.

[0012] In addition to the above-described map, in accordance with the invention, a so-called environment model is defined which encapsulates a minimal set of assumptions that must be made about the particular operating environment in which the event driven application is executed. More particularly, as the model checker input language is provided to the model checker, the environmental model is applied. As a result, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to the environmental model during the testing and checking of the event driven software by the verification system. Thereafter, the testing and checking results are output, in a well-known manner, by the model checker for determining whether the event driven system conforms with the user's desired execution properties and behaviors.

[0013] Advantageously, in accordance with the invention, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the event driven system program code without the need for user intervention.

Brief Description of the Drawings

[0014]

FIG. 1 is a block diagram of an illustrative event driven system;
FIG.'s 2 and 3 show an example of a conventional C source code program for an illustrative telecommunications call processing application useful in the illustrative event driven system of FIG. 1;
FIG. 4 shows a flowchart of illustrative operations for software development in accordance with the principles of the invention;
FIG.'s 5 and 6 show an illustrative C source code program, structured and defined in accordance with the software development operations of FIG. 4, for the illustrative telecommunications call processing application of FIG.'s 2 and 3;
FIG.'s 7 and 8 show a translated source code program derived, in accordance with the principles of the invention, from the illustrative C source code program of FIG.'s 5 and 6;
FIG. 9 shows a flowchart of illustrative operations for testing event driven software in accordance with the present invention;
FIG.10 shows intermediate state machine code derived from the illustrative C source code program of FIG.'s 5 and 6;
FIG. 11 shows an illustrative map for the event driven system defined by the illustrative C source code program of FIG.'s 5 and 6;
FIG.'s 12 and 13 show model checker input language derived from applying the illustrative map of FIG. 11;
FIG. 14. shows an illustrative set of library functions for use with the model checker input language of FIG.'s 12 and 13;
FIG. 15 shows an illustrative environmental model for the telecommunications processing application described in FIG.'s 5 and 6;
FIG. 16 shows an illustrative automaton model for a particular control state defined by the telecommunications processing application of FIG.'s 5 and 6;
FIG. 17 shows an illustrative full automaton model of the complete telecommunications processing application described in FIG.'s 5 and 6;
FIG. 18 shows an illustrative computer system 1800 useful in executing event driven software code which is specified and written in accordance with the present invention; and
FIG. 19 shows an illustrative error trace, generated in accordance with the principles of the invention, for the telecommunications processing application described in FIG.'s 5 and 6.
Throughout this disclosure, unless otherwise noted, like elements, blocks, components or sections in the figures are denoted by the same reference designations.

Detailed Description

[0015] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all other rights with respect to the copyrighted works whatsoever.

[0016] The present invention provides a technique for testing event driven software. In accordance with various aspects of the invention, the event driven system program source code is directly converted to a automaton based model useful in verifying that the program code complies with the desired properties defined by the user. More particularly, in accordance with a preferred embodiment of the invention, the event driven system program code is translated into a target language for a particular model checker. Such a translation results in a model which contains statements directed at whether execution of the program code will affect the behavior of the event driven system, e.g., the execution of the program code causes new events to be generated. Thus, this model extraction process of the preferred embodiment of the invention can be used as input to a logic model checker for determining whether event driven system complies with the desired correctness properties specified by the user. Advantageously, in accordance with the invention, the model extraction process and application of the model checker occurs in a direct and dynamic fashion from the subject event driven system program code without the need for user intervention.

[0017] The present invention can be embodied in the form of methods and apparatuses for practicing those methods. The invention can also be embodied in the form of program code embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. The invention can also be embodied in the form of program code, for example, in a storage medium, loaded into and/

or executed by a machine, or transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via electromagnetic radiation, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. When implemented on a general-purpose processor, the program code segments combine with the processor to provide a unique device that operates analogously to specific logic circuits.

[0018] In order to provide context and facilitate an understanding of the invention, an overview and discussion of an illustrative event driven system will now be presented. In particular, FIG. 1 shows a block diagram of an illustrative communications network which, as will appreciated, contains a myriad of event driven applications. Communications network 100, e.g., is a public switched telephone network such as the well-known inter-exchange network of AT&T Corp., that provides long distance telephone services for its subscribers. These subscribers access communications network 100 through, e.g., communications devices 105-1 through 105-6, which are, e.g., customer premise equipment, wired telephones, personal computers, cellular telephones, pagers and facsimile machines. Communications network 100 includes, *inter alia,* a plurality of toll switches, e.g., shown in FIG. 1 as toll switches 115, 120, 125, and 130. These toll switches may be any of the well-known types of telecommunications switching equipment, e.g., the No. 4ESS® (Electronic Switching System) or the No. 5ESS® available from Lucent Technologies Inc., 600 Mountain Ave., Murray Hill, NJ 07974. As shown in FIG. 1, each of toll switches 115, 120, 125, and 130 are connected to a number of other switches via a so-called inter-toll network shown as block 150. Each toll switch may also be connected to multiple central offices ("CO"), e.g., CO's 110 through 114. The operation of such telecommunications networks and CO's is well-known, e.g., as discussed in "Engineering and Operations in the Bell System", Second Edition, Eighth Printing, International Standard Book Number 0-932764-04-5, 1993, and the detail of which will not be further discussed herein. In short, a CO is arranged to extend to a corresponding toll switch of communications network 100 a telephone call originating at, e.g., communications device 105-1, from which a calling party has dialed a particular telephone number. The CO, e.g., CO 110, is further arranged to extend the call connection to, e.g., communications device 105-6 associated with the called party and to the CO, e.g., CO 114, which receives the extension of the call from the corresponding toll switch, e.g., toll switch 125.

[0019] Toll switches 115, 120, 125 and 130 of communications network 100 are interconnected via data link 135, which may be, e.g., the well-known System Signaling 7 ("SS7") network. Communications network 100 is arranged so that the toll switches may exchange data messages with one another to establish a connection between a calling party (e.g., communications device 105-1) and a called party (e.g., communications device 105-6), via communications network 100. That is, the connection is made by extending a call through communications network 100 until the call is completed (e.g., the called party answers the call by going "off-hook") between the calling party and the called party. Communications network 100 further includes a number of centralized databases commonly known as Network Control Points ("NCPs"), a single one of which is shown as NCP 140. As is well-known, NCP's, such as NCP 140, are strategically positioned at various positions within communications network 100 to support various service features accessed and provided through the network such as the well-known "800" or "888" toll-free telephone number services.

[0020] As will be appreciated, communications network 100 and the various above-described elements of the network require a myriad of event driven applications. For example, call processing application programs for toll switches 115, 120, 125 and 130 must be developed to facilitate the extension of a call through network 100. In particular, for example, a calling party may wish to utilize communications device 105-1 for placing an outgoing call. Such an event driven call-processing application will include operations, i.e., control states, such as offhook, dial, idle, connect, ring, and busy, to name just a few.

[0021] As mentioned above, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the previously cited, co-pending, commonly assigned patent application, entitled "Method and Apparatus For Developing Event Driven Software" (hereinafter referred to as "the Method and Apparatus For Developing Event Driven Software application"). Thus, in order to further facilitate a complete understanding of the present invention, the various aspects of the invention in "the Method and Apparatus For Developing Event Driven Software application" will be now be discussed in some detail.

[0022] The various aspects of the invention in that application, provide a programming technique for defining, structuring, and developing event driven software. More particularly, event driven software is structured in two basic parts which are referred to therein and herein specifically as control-sensitive code and non-control-sensitive code. In accordance with the invention, control sensitive code includes the identification of control states, the events that are generated and that must be recognized, and the responses to the arrival of particular events in particular states. Further, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. The non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc. Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly

impacting the behavior of the event driven system. As such, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself.

[0023] More particularly, FIG. 2 and FIG. 3 show an example of a conventional C source code program 200 (delineated by individual program code line numbers 1-268) directed to such an illustrative event driven application for processing an outgoing call. As will be appreciated, source code program 200 defines several control states for the application. See, for example, code sections 205 through 230 which define operations for the following respective control states: busy, conn, dead, digits, idle, and ring. Significantly, however, an examination of source code program 200 does not provide an immediately evident view of the execution thread between the various control states. From a high level programming perspective , there is no direct or logical, i.e. causal, relationship between the various control states defined within the source program code. That is, as mentioned above, in order for a programmer to understand the logical flow from control state-to-control state directly from the program source code it is necessary to trace the execution thread through the source, e.g., tracing branch instructions or "goto" statements. As will appreciated, the tracing of such logical flow, particularly in very large source code programs, is a tedious, unilluminating, error prone, and time consuming task. For example, the source code for implementing all the requisite call processing features of communications network 100 will most likely be on the order of tens of thousands lines of code. This also leads to certain complexity in the testing of the software for so-called "bugs" and to determine whether the application program implements the desired features. Significantly, the novel programming technique is directed to defining, structuring, and coding event driven software which mitigates certain disadvantages of the prior art. In accordance with the invention, control sensitive code includes the identification of control states, the events that are generated and that must be recognized, and the responses to the arrival of particular events in particular states. Further, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. The non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc.

[0024] Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. As such, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself.

[0025] In accordance with the preferred embodiment of the invention, a programming extension to the well-known ANSI-standard C programming language is adopted for facilitating the definition, structure and coding of event driven software in accordance with the invention. More particularly, in accordance with the preferred embodiment of the invention, the extension to the ANSI-standard C programming language consists of a single symbol, illustratively chosen to be the well-known ASCII symbol "@", and used for labeling all control states within the control sensitive portion of the program code. That is, the symbol "@" is used as the initial character of all labels that correspond to control states within the program source code. Thus, in accordance with the invention, a program statement prefixed with the "@" label identifies a control state in the program code whereby the current execution of the event driven system waits for the arrival of the next event. Following the arrival of the next event, execution of the subsequent program code follows along with the encoding of the required response to that event, and concluding with a transition to the new, uniquely defined control state. As will be appreciated, the response and the new control state, can be different for each event being processed depending upon operational issues such any state information stored in internal tables, variables, device responses, and the like.

[0026] FIG. 4 shows a flowchart of illustrative operations for software development in accordance with the principles of the invention. More particularly, a specification for the event driven system to be programmed is defined (block 410), e.g., by the system engineer. As will be appreciated by those skilled in the art, such a specification provides a description of the overall application, e.g., event driven system, for which a software program is to be written and implemented. From the system specification, the control states for implementing the event driven system under development can be identified (block 420) which, in turn, leads to the identification of the events which will be generated and that must be recognized, and the responses to the arrival of particular events in particular states. As such, in accordance with the invention, the definition, structuring, and coding (block 430) of the source code program is divided into two parts, as discussed above, the control-sensitive code and the non-control-sensitive-code.

[0027] In accordance with the invention, within the control-sensitive code defining the system, control states are uniquely identified throughout such program code with a single symbol. As discussed above, in accordance with the

preferred embodiment of the invention, the symbol "@" is used as the initial character of all labels that correspond to control states within the control-sensitive portion of the overall program source code. Thus, in accordance with the invention, a program statement prefixed with the "@" label identifies a wait-state in the program code, where the current execution of the event driven system waits for the arrival of the next event. Further, the non-control-sensitive code, specified separately from the control sensitive code, is directed at the necessary normal sequential code for programming the event driven system such as the invocation of device drivers, the control of peripheral devices, the updating of tables, variable definition, etc.

[0028] Advantageously, the invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. Thus, one practical result of applying the principles of the invention is the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context. That is, in accordance with the invention, the program source code is structured and defined such that the control states which define the event driven application have a direct and logical, i.e. causal, relationship which is immediately evident from an examination of the source code itself. Significantly, our present invention leads to further advantages in the testing and verifying of the event driven software (block 440) to determine if the software satisfies the desired properties as specified by the user. In particular, it is the testing of such event driven software that the present invention is directed and further discussed below in greater detail.

[0029] However, continuing with the discussion of "the Method and Apparatus For Developing Event Driven Software application", as mentioned above, that invention allows the software developer to structure the event driven software application code such that a clear distinction is made between core control features, i.e., states, events and responses, and extraneous details not directly impacting the behavior of the event driven system. For example, FIG.'s 5 and 6 show an illustrative C source code program 500 (delineated by individual program code line numbers 1-258), structured and defined in accordance with the invention, for the illustrative telecommunications call processing application of FIG.'s 2 and 3. In particular, control states 510 through 555 are delineated, in accordance with the preferred embodiment, with the symbol "@"as the initial character of all labels thereby designating the control states within the control-sensitive portion of the overall source code program 500. Thus, source code program 500, in accordance with the invention, has a direct and logical, i.e. causal, relationship between the individual control states, i.e., control states 510-555. Significantly, the programmer can determine the exact execution flow from control state-to-control state directly from a cursory examination of the program code itself as the individual control states have a strong visual context.

[0030] For example, through a cursory examination of source code program 500, one can quickly and exactly ascertain when viewing the code at control state 535, i.e., the "@digits" control state, how that particular control state is reached. That is, invocation and execution of this particular event driven system, i.e., the processing of an outgoing call, a control state for allowing the dialing of digits on a telephone device is processed. Control state 535 is such a state and is clearly identifiable beginning at line 125 of code 500, in accordance with preferred embodiment of the invention. In viewing control state 535, one can readily see that the execution flow in reaching this control state is directly through the above-coded control states 510 through 530, respectively. Further, if source code program 500 needs to be revised to add another control state, e.g., to add some further application-specific functionality, the new control state can be placed at the exact location within the existing control state hierarchy without losing context between all the control states.

[0031] This logical, i.e. causal, relationship among control states and the clear execution flow directly perceived from viewing source code program 500 is further evident in FIG.'s 7 and 8 which show a translated source code program 700 derived from the illustrative C source code program 500 of FIG.'s 5 and 6. As mentioned above, in accordance with the preferred embodiment of the invention, the extension to the ANSI-standard C programming language consists of a single symbol illustratively chosen to be the symbol "@" and used for labeling all control states within the control sensitive portion of the program code. The mechanics, i.e. the translator, of the actual translation of the code itself will be readily apparent to those skilled in the art. Thus, translated source code program 700 is the actual fully translated ANSI-standard C representation of the event driven application program in accordance with the invention. For example, control state 515 (i.e., "@idle") of FIG. 5 is translated to program code section 710, and control state 545 (i.e., "@ring") is translated to program code section 720.

[0032] The above-described advantages of the invention in "the Method and Apparatus For Developing Event Driven Software application" provide significant utility in the computer programming arts, in particular, to the definition, structure, and development of event driven software application programs. Of course, critical to the successful development of any application program is the effective testing and verification of the program code against a defined set of expected properties. It is the testing of event driven software that the various aspects of the present invention are directed and which now will be discussed in greater detail.

[0033] The various aspects of the present invention are directed to the efficient testing of event driven software and making it possible to track the validity of future changes to the program code, as part of the normal maintenance,

extension and revision process to the source code. FIG. 9 shows a flowchart of illustrative operations for testing event driven software in accordance with the principles of the invention. More particularly, in accordance with the preferred embodiment of the invention, event driven software code is specified and written in accordance with the principles of the invention in "the Method and Apparatus For Developing Event Driven Software application", as described previously. That is, in accordance with the preferred embodiment, the source code program is defined and structured into control-sensitive and non-control-sensitive format (block 910.) In accordance with the invention, the event driven source code program is parsed to extract the control states defined in the source program, and to convert the source code into an intermediate state machine format (block 920.) Thereafter, in accordance with the invention, the intermediate state machine format is converted into a automata-based format for modeling checking purposes.

[0034] In particular, in accordance with the preferred embodiment, the intermediate state machine format is converted into an annotated automaton model in the input language of a specific model checker (block 940). In accordance with the preferred embodiment, the model checker tool is the well-known "SPIN" model checker developed by and available from the Bell Laboratories Division of Lucent Technologies Inc., and as described in more detail, e.g., in G. J. Holzmann, The Model Checker SPIN, *IEEE Trans. On Software Engineering,* Vol. 23, No. 5, pp. 279-295, May 1997, which is hereby incorporated by reference for all purposes. Importantly, in accordance with the invention, each statement from the source code program is mapped (block 930) onto an abstraction within the automaton model. The mapping, discussed below in greater detail, is facilitated by a translation map which is defined at the initial stage of the verification process. The map need not be revised thereafter unless new types of instructions are introduced, e.g., after revisions are made to repair programming faults, into the source program. The mapping, *inter alia,* dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the source code.

[0035] In addition to the above-described map, in accordance with the invention, a so-called environment model is defined, illustratively by a user, which encapsulates a minimal set of assumptions that must be made about the particular operating environment in which the event driven application is executed. More particularly, as the model checker input language is provided (block 950) to the model checker, e.g., SPIN, the environmental model is applied (block 975). As a result, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to the environmental model during the testing and checking (block 960) of the event driven software by the model checker. Thereafter, the testing and checking results are output, in a well-known manner, by the model checker (block 970) for determining whether the event driven system conforms with the user's desired execution properties and behaviors.

[0036] Important to the above-described testing and checking of event driven software, in accordance with the invention, is the efficient parsing of the event driven source code program to extract the control states defined in the source program, and converting the source code into an intermediate state machine format (see, e.g., FIG. 9, block 920.) That is, to perform model checking efficiently, one must be able to identify the control states, events and actions (i.e., responses) of the event driven system. Thus, a further aspect of the invention is directed to our realization of a formal specification which facilitates the description of an event driven system for model checking purposes, as will now be discussed.

[0037] In accordance with this aspect of the invention, a format is defined which facilitates the specification of an event driven system. In accordance with various embodiments of the invention (discussed further below), the specification may be converted into a target programming language, e.g. C., for direct compilation and execution. In accordance with still further embodiments of the invention, the specification is converted, using the above-described mapping, into a logical verification model such as the input language of the SPIN model checking tool. As will be appreciated by those skilled in the art, grammar specification tools are well-known. For example, the YACC (yet-another-compiler-compiler) is one well-known UNIX software utility tool (see, e.g., A. T. Schreiner, Using C with *curses, lex* and *yacc,* Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1990) which assists programmers in the development of C routines which analyze and interpret an input stream, and also facilitates the development of compilers and interpreters. Similar to YACC, our specification format provides a formalism for describing the grammar of a language, and automating the generation of a compiler for that language. However, our specification format, in accordance with the invention, is directed towards the specification of the behavior of the system.

[0038] More particularly, an illustrative grammar definition, in accordance with the preferred embodiment of the invention is as follows:

```
spec:   defs "%%" rules
        ;
defs:   /*empty*/
        |       one_def
        |       one_def defs
        ;
one_def:        "%c_template" "\" " STRING "\""
        |       "%p_template" "\" " STRING "\""
        |       "%p_map" "\" "STRING "\""
        |       "%nparam"NUMBER
        |       "%event" names
        |       "%state" names
        ;

 names:         STRING
        |       STRING names
        ;

rules: one_rule
        |       one_rule rules
        ;

one_rule:       source ":" opt_recpt responses ";"
        ;

responses:      one_response
        |       one_response "|" responses
one_response:           triggers opt_target opt_action
        ;
```

```
triggers:       one_trigger
        |       one_trigger "+" triggers
        ;
one_trigger:event
        |       "("condition")"
        |       "!("condition")"
        |       "else"
        |       "always"
        ;
opt_target:     /*empty*/
        |       target
        ;
opt_action      /*empty*/
        |       "{"action"}"
        ;
opt_recpt:      /*empty*/
        |       onreceipt"{"action"}"
        ;

event:          STRING;
condition:      STRING;
action:         STRING;
source:         STRING;
target:         STRING | "@" STRING;
```

The above illustrative grammar definition provides the framework for defining a specification for a particular application, e.g., an event driven system, in accordance with the principles of the invention. All items in the illustrative grammar definition which appear in double quotes are terminal symbols (keywords, literals, tokens). The capitalized word STRING represents an alphanumeric string, and NUMBER represents a sequence of one or more digits. Alternatives are separated by a vertical bar "|", i.e., pipe symbol, and each grammar rule is terminated by a semicolon.

[0039] In accordance with the preferred embodiment, the specification format consists of two parts: a declaration and a specification. The declaration part of the specification of the preferred embodiment, contains, *inter alia,* a listing of symbolic names for all events that will be used in the specification. For example, the following code fragment is an

illustrative declaration in accordance with the preferred embodiment:

```
%c_template      "c.ctp"  /*optional: a C code template */
%p_template      "p.ctp"  /*optional: a SPIN model template */
%p_map           "p.map"  /* optional: maps C code to SPIN */

%event Cranswer
%event Crbusy
%event Crconn
%event Crdigit

%state S_idle
%state S_pots
%state S_alloc
%state S_parked
%state S_busy
%%
```

In the above illustrative declaration, all words that begin with a percent "%" symbol are keywords. For example, "% c_template" defines the name of a file that contains a template for the final implementation of the code in C, i.e., the target implementation language of the preferred embodiment. Further, the "%p_map" statement gives the name of a file that can be used to define a relation between the C code from the implementation and the abstract representations of the code, i.e., mapping, used for verification with the model checker. In addition, the illustrative declaration above further contains a listing of symbolic names for all events, e.g., "%event Cranswer", that will be used in the specification, as well as, a list of state names for all of the control states, e.g., "%state S_idle", in the specification.

[0040] The core of the specification format, in accordance with the invention, is the specification of a set of transition rules for the event driven system wherein the state specification consists of a sequence of transition rules specified in the following notation:

```
state   : event1 targetstate1 {code fragment1}
        | event2 targetstate2 {code fragment2}
        .....etc.
        ;
```

For example, the following code fragment is an illustrative control state specification in accordance with the preferred embodiment:

```
S_idle : Cronhook S_idle
             {x→cwswit &= Swhook;}
       | Crflash
             {x→cwswit |= Swhook;}
       | Croffhook @S_idle_E3
             {x→cwswit |= Swhook;}
       | Iring @S_idle_E5
       ;
```

In the above illustrative control state specification, the transition rule begins with the name of the state, e.g., "S_idle", followed by a colon. After the colon, a series of possible transitions out of this state are listed. The individual transitions are separated from each other by a vertical bar "|". Each complete transition has three parts: the name of an event that triggers the response, the name of a new state that is reached after the response has been performed, i.e., "the next-state", and the response itself. The next-state part of the specification defines where control moves after the response has been completely processed, and contains the name of either a predefined control state or a so-called internal state. If two events have the same next-state and response, in accordance with further embodiments of the invention, an abbreviated notation can used such as:

```
       | Crflash
       + Crdis
       + Croffhook S_idle
             {x→cwsit |= Swhook;}
```

Using the above noted abbreviated notation, the events Crflash, Crdis, and Croffhook will all generate the same response, i.e., the execution of the specified code fragment and a transition to the state "S_idle".

[0041] As will be appreciated, the event driven system is meant to stop execution at each control state and wait for the arrival of the next event. In contrast, the event driven system passes through internal states without waiting. In addition, state names in the next-state segment of a transition rule can be prefixed with the "@" symbol (it should be noted that this particular notation is not the same as that described previously with regard to identifying control states

in accordance with the various aspects of the invention in "the Method and Apparatus For Developing Event Driven Software application") to indicate that processing of the next state should continue without waiting for a next event to arrive. In such a case, the last event remains available if an event is needed to determine the behavior of the system at a next state.

[0042] With reference to the internal states mentioned above, such events encode an internal, non-event related decision of the event driven system. In accordance with the preferred embodiment of the invention, the syntax in the specification for internal events is the same as for control states, except that instead of event names, general Boolean conditions in the target implementation language are used. For example, an illustrative internal state is defined as follows:

```
S_idle_E5:    {x→drv→trunk} @S_otrunk
                    {x→orig=ipc→orig;
                        ipc→orig→term = x;
                        emitcdr{x,CDRtransit,0};}
            | ! {x→drv→trunk} @S_idle_E6
            ;
```

The above illustrative internal state specifies that the value of a specific field (in this case hiding two levels inside a C data structure named "x") will decide whether control moves into state "S_otrunk" or state "S_idle_E6". In the first case, the piece of action code defined after the state will be executed. In the latter case, no code is executed.

[0043] In accordance with the preferred embodiment, if the name of an event or condition is "else" this indicates that a default response will be taken when none of the explicitly listed event names or Boolean conditions apply. For example, the above illustrative internal state code fragment can also be written as follows:

```
S_idle_E5:    {x→drv→trunk} @S_otrunk
                    {x→orig=ipc→orig;
                        ipc→orig→term = x;
                        emitcdr{x,CDRtransit,0};}
            | else @S_idle_E6
            ;
```

In accordance with this alternative internal state code fragment, no code fragment is executed when the condition "{x→drv→trunk}" evaluates to "false" upon reaching this state. Instead, only a transition into state "S_idle_E6" will result.

[0044] Further, with regard to the specification of the event driven system, it is sometimes useful to define a common action fragment that must be executed immediately following the reception of an event in a control state. By definition,

such a code fragment is to be executed before any of the transition rules are applied. In accordance with the preferred embodiment of the invention, the keyword "onreceipt" is employed to define such common action fragments. An illustrative example of this keyword's use is as follows:

```
S_orig : onreceipt
        {x→cwswitch&= ~Swconn;
            y=callroute(x→endpt→ctxt, x→lidb→number,&i);}
    | (y = = nil) @S_orig_E54
    | else @S_orig_E55
    ;
```

In the above illustrative code fragment, the state is an internal state so it can always be referred to in a next-state segment with the "@" prefix. The "onreceipt" code is immediately executed when reaching this state on a transition, and the condition "(y = = nil" is evaluated. If the condition is "true", the system moves into another internal state named "S_orig_E54", and if "false", the system move into internal state "S_orig_E55" where processing continues without waiting for a new event.

[0045] Finally, with regard to the specification, it is sometimes useful to bypass condition checks altogether and specify an intermediate transition state that will lead unconditionally to a successor state. In accordance with the preferred embodiment of the invention, a condition that is always true is represented by the keyword "always" as is illustrated in the following example code fragment:

```
S_enabdial    : onreceipt
            {x→ndigits=0;
            x→drv→dtmf(x,1);
            x→cwswit |= Swconn;}

    | always S_dial
    ;
```

[0046] Our realization of the above-described formal specification in describing event driven systems for model checking purposes advantageously provides a specification format for describing the behavior of a event driven system.

[0047] To further illustrate the various aspects and advantages of the invention in testing event driven software, FIG. 10 shows intermediate state machine code 1000 (delineated by individual program code line numbers 1-124) derived, in accordance with the principles of the invention as discussed above, from the source code listing of the illustrative C source code program 500 of FIG.'s 5 and 6. As discussed above, to effectively perform model checking on event driven code, an accurate assessment and identification of the control states, events, and actions must occur. Advantageously, intermediate state machine code 1000, in accordance with the principles of the invention, provides a clear delineation amongst control states, events and actions. More particularly, in accordance with the preferred embodiment, a translator is applied to C source code program 500 to directly generate intermediate state machine code 1000. The actual construction of the translator will be readily understood by those skilled in the computer programming art and need not be discussed further herein. However, by structuring C source code program 500 in accordance with the present invention, the automatic identification of the relevant control states, events, and actions is made possible. Thus, the advantages of the present invention as set forth herein are realized.

[0048] More particularly, a careful examination of intermediate state machine code 1000 show a clear delineation of events (see, e.g., code section 1010) and control states (see, e.g., code section 1020) of the declaration part of code 1000. Further, the clear interaction amongst and between such events, control states and the related actions is shown, in accordance with the invention, by the various state specifications (see, e.g., state specifications 1030 through 1050) within intermediate state machine code 1000. As discussed above with reference to the operations of FIG. 9, in accordance with the invention, intermediate state machine code 1000 is converted to the model checker input language for executing the testing of the subject program code. This conversion is facilitated by a translation map which is defined at the initial stage of the verification process. The mapping, *inter alia,* dictates a fixed translation of the particular statement into the target language of the model checker. Thus, instructions that appear in the map are systematically converted into the automaton model wherever such instructions appear in the original source code.

[0049] FIG. 11 shows an illustrative map 1100 for the event driven system defined by the illustrative C source code program 500. Map 1100 consists of program code 1110 (delineated by individual program code line numbers 1-47) which defines how the fixed translation of particular statements in the original source code is made into the target language of the model checker. In particular, program code 1110 is includes event definitions 1120 and code definitions 1130. For example, line 20 of program code 1110 maps every occurrence of the statement in the left hand column onto the abstraction given in the right hand column. In accordance with the invention, the application of map 1100 results in the derivation of model checker input language 1200 (delineated by individual program code line numbers 1-303) shown in FIG.'s 12 and 13. Further, in accordance with the invention, a set of library functions is generated, illustratively shown as library functions 1400 in FIG. 14, for the model checker input language of FIG.'s 12 and 13. As will be appreciated, library functions 1400 (delineated by individual program code line numbers 1-153) provide the necessary semantics for defining terms in model checker input language 1200. In accordance with the preferred embodiment, model checker input language 1200 is directed to and useful with the SPIN model checker tool.

[0050] Significantly, one important feature of using maps, in accordance with the invention, is the capability provided to the user in defining precisely the appropriate level of detail at which the checking process will be applied, in a source independent manner. The map can contain "true" or "skip" as the translation (i.e., right hand column of program code 1110 described above) of any statement which indicates that this detail is abstracted away. Therefore, the map acts as a "filter" to filter out extraneous detail, in addition to, acting as a converter from the target programming language, e.g., C, to the verification model. The map also serves as a complete formalization of the linkage between source code and the verification model.

[0051] As described above, in accordance with the preferred embodiment of the invention, the verification of the properties of the event driven system is made subject to an environmental model during the testing and checking (see, FIG. 9, block 975) of the event driven software by the verification system. The environmental model formalizes particular assumptions, made by the user, about the environment in which the event driven program code is to be executed. For example, in the case of the illustrative telecommunications call processing application described herein, the environmental model might include formalizations directed to subscriber behavior, hardware responses, and communications network performance. In particular, FIG. 15 shows an illustrative environmental model 1500 (delineated by individual program code line numbers 1-80) for the telecommunications processing application described in FIG.'s 5 and 6. For example, code fragment 1510 of environmental model 1500 describes a formalization, i.e. a template, for a telecommunications device useful in the illustrative telecommunications application. In accordance with the preferred embodiment, the model checker, i.e., SPIN, will use model checker input language 1200 in conjunction with environmental model 1500 to determine whether the event driven system conforms with the user's desired execution properties and behaviors.

[0052] Advantageously, the separate definition of the map and environmental model, in accordance with the invention, ensure that little, if anything, in the verification infrastructure needs to be updated when routine changes are made in the original source code. That is, unless new basic types of statements are introduced in the source code, the verification process can be repeated without user intervention and the source code may be checked for continued

compliance against a large library of essential correctness properties. Further, in accordance with the invention, a precise automaton model representative of the event driven code under test is automatically generated directly from the source code and verified with a logic model checker, e.g., SPIN.

[0053] For example, FIG. 16 shows an illustrative automaton model 1600 for a particular control state defined by the telecommunications processing application described in FIG.'s 5 and 6. More particularly, the control state depicted in FIG. 16 is the so-called "ring" control state defined in source code program 500 as control state 545 (see, FIG.'s 5 and 6). The "@ring" program statement in line 183 of source code program 500 begins the clear delineation of this particular control state from a source program viewpoint. In accordance with the testing and checking aspects of the invention, as described above, from this control state a precise automaton, e.g., automaton model 1600, is extracted. Automaton model 1600 clearly shows the transitions in and out of the "ring" control state 1610 and amongst other control states of the application, i.e., "digits_1" control state 1620, "idle" control state 1630, "conn" control state 1640, and "error" control state 1650. Automaton model 1600 also clearly illustrates the various events, i.e., event 1660 through event 1695, occurring in the illustrative application. Further, full automaton model 1700, shown in FIG. 17, is an illustrative full automaton model of the complete telecommunications processing application described in FIG.'s 5 and 6, and automatically generated in accordance with the invention.

[0054] The above-described embodiments of the invention employ a C source program written (see, e.g., FIG.'s 5 and 6) in accordance with the principles of the "the Method and Apparatus For Developing Event Driven Software application", and tested in accordance with the principles of the present invention as described above. We have further realized that certain types of users, e.g. verification system programmers or software testing engineers, may wish to begin their system design process by defining the event driven in accordance with the specification format described above. For example, such users may employ the specification format set forth herein to directly code the subject event driven system. Thus, in accordance with further embodiments of the invention, the specification is converted into a target programming language, e.g. C., for direct compilation and execution. In accordance with such further embodiments of the invention, a template is employed to facilitate such a conversion. The following is an illustrative C code template for use in converting this such state machine code to standard C for compilation and execution:

```
void
state(Line *x, Ipc *ipc)
{       Line *y;
        int op, i, c, os, time;

        goto start;
```

```
        /*** template: ***/
        @C

    out:
        if (x == nil)
    return;
        if (time != 0)
        {
        x->time=rtms()+time;
        addtimer(x);
        }
        return; /* exit-point */
    start:
        op=ipc->type; os=x->state; time=0; switch (os)
        {
        @@
        }
        os=0;
    error:
        y=x->term; x->term=nil; goto Aerror;
    }
```

[0055] The above-described C code template provides the framework for a complete conversion of the state machine specification to C. As will be appreciated, the illustrative C code template can be executed by the model checker tool itself or on a separate machine. The template includes type definitions for the arguments that are passed to a routine called "state" that is used as the core of the conversion implementation. The routine must be called whenever an event occurs that relates to the subject event driven system. The template code further contains a place-holder "@C" for the state machine that is generated from the transition rules specified in accordance with the invention. The actual program code necessary for the generation of the state machine will be readily understood by those skilled in the art and need not be further detailed herein. At each call, the "start" routine executes and retrieves an event type and parameter value from the data structure that is used by the environment to store detailed information about the current event. The current state of the process is retrieved from the data structure in the first argument to the routine, and a so-called "C switch" is performed on this state value to establish the proper position in the transition rules. The short-hand symbol "@@" of the template is expanded by the tool into a full list of names of all control states, with matching jumps to the places in the generated, i.e., converted, C code where the code fragments for the transition rules of each state are

placed. Advantageously, in accordance with these further embodiments of the invention, the specification is converted into a target programming language, e.g., C, for direct compilation and execution.

[0056] For example, FIG. 18 shows an illustrative computer system 1800 useful in executing and testing event driven software code in accordance with the present invention. In particular, computer system 1800 includes conventional personal computer 1810 which is connected to display 1820 and keyboard 1830. As will be appreciated, information produced by personal computer 1810, e.g., during execution of particular event driven software code, may be displayed on display 1820 for access by a user of computer system 1800. The user, as a result of viewing display 1820, interacts with and controls personal computer 1810, in a conventional manner, using keyboard 1830. Further, mouse 1880, or other well-known pointing devices, may allow the user to provide additional inputs to personal computer 1810. Illustrative computer system 1800 further includes processor 1840 which, illustratively, executes and tests event driven application software code in accordance with the present invention. For example, such event driven application software code may be loaded into personal computer 1810 via conventional diskette 1870 and thereafter stored in internal disk memory 1860. Further, execution and testing of the code may require access and use of random access memory 1850 ("RAM") in a conventional manner. Thus, the user of computer system 1800 can execute and test software, in accordance with the present invention, to provide a wide variety of applications, e.g., an event driven system. Further, as will be appreciated, computer system 1800 may also be connected, in a conventional manner, to other computer systems or devices for execution and testing of the particular application programs.

[0057] For example, computer system 1800 can be used to execute and test software, in accordance with the present invention, resulting in error trace 1900 as shown in FIG. 19. Error trace 1900 consists of program code 1910 (delineated by individual program code line numbers 1-47) and is an illustrative error trace, generated in accordance with the principles of the invention, for the telecommunications processing application described in FIG.'s 5 and 6. More particularly, the illustrative error trace 1900 shows how the checking process produces a scenario which directly identifies the presence of errors in the original source code of the event driven system. Further, error trace 1900 through program code 1910 identifies such errors in terms of programming level statements, e.g., C-level statements, that may be executed to reproduce the error. In the illustrative error trace 1900, the trace demonstrates in a direct sequence of C-program code statement (see, FIG. 9, program code 1920) executions that leads the event driven system into the illustrate state "@busy" with "Crdigit" being the first event to be processed. As can be seen and appreciated from error trace 1900, in conjunction with, C source program code 500 (in particular, control state 555 at line 223), there is no provision in the program code, i.e., a coding error, to anticipate the possible occurrence of this event. Advantageously, in accordance with the invention, a user can interpret error trace 1900 solely in terms of C-level execution statements without any specialized knowledge of the model itself or significant parts of the intermediate representations that are used in the checking process itself.

[0058] The foregoing merely illustrates the principles of the present invention. It will thus be appreciated that those skilled in the art will be able to devise various arrangements which, although not explicitly described or shown herein, embody the principles of the invention and are included within its scope. Furthermore, all examples and conditional language recited herein are principally intended expressly to be only for pedagogical purposes to aid the reader in understanding the principles of the invention and the concepts contributed by the Applicant(s) to furthering the art, and are to be construed as being without limitation to such specifically recited examples and conditions. Moreover, all statements herein reciting principles, aspects, and embodiments of the invention, as well as specific examples thereof, are intended to encompass both structural and functional equivalents thereof. Additionally, it is intended that such equivalents include both currently known equivalents as well as equivalents developed in the future, i.e., any elements developed that perform the same function, regardless of structure.

[0059] Thus, for example, it will be appreciated by those skilled in the art that the block diagrams herein represent conceptual views of illustrative circuitry embodying the principles of the invention. Similarly, it will be appreciated that any flowcharts, flow diagrams, state transition diagrams, pseudocode, program code, and the like represent various processes which may be substantially represented in computer readable medium and so executed by a computer, machine, or processor, whether or not such computer, machine, or processor, is explicitly shown.

[0060] Further, in the claims hereof any element expressed as a means for performing a specified function is intended to encompass any way of performing that function, including, for example, a) a combination of circuit elements which performs that function; or b) software in any form (including, therefore, firmware, object code, microcode or the like) combined with appropriate circuitry for executing that software to perform the function.

Claims

1. A software testing method, carried out on a source program of the software, the source program including a plurality of instructions, said method comprising:

parsing the source program to identify a plurality of control states and converting the source program to a state machine format;
applying a translation map to the plurality of instructions of the source program;
converting the state machine format of the source program to a verification program;
applying a environmental model to the verification program; and
testing the software in a verification tool using the verification program.

2. The software testing method of claim 1 wherein the source program contains a first source code segment for causing a computer to execute a event driven system; the first source code segment including a first plurality of instructions defining the plurality of control states, the plurality of control states being associated with the event driven system, a second plurality of instructions defining a plurality of events of the event driven system, and a third plurality of instructions defining a plurality of actions of the event driven system, each control state of the plurality of control states having a individual identifier within the first plurality of instructions of the first source code segment, the individual identifier being a programming symbol which is common to all the identified control states, and a second source code segment containing at least one plurality of instructions defining a sequential control of the computer executing the event driven system.

3. The software testing method of claim 1 or claim 2 wherein the translation map includes a plurality of mapping instructions.

4. The software testing method of claim 3 wherein the applying the translation map operation includes:

comparing the plurality of mapping instructions to the plurality of instructions of the source program to detect a match; and
converting particular ones of the mapping instructions which match particular ones of the instructions of the source program to the input language of the model checker.

5. The software testing method of any of the preceding claims wherein the environmental model includes a plurality operational attributes of the event driven system.

6. The software testing method of any of the preceding claims wherein the verification tool is a model checker tool.

7. The software testing method of claim 6 wherein the model checker tool is a SPIN model checker tool.

8. The software testing method of any of the preceding claims wherein the event driven system is a telecommunications service application, the telecommunications service application having at least one event associated with extending a telephone call through a communications network.

9. The software testing method of any of the preceding claims wherein the state machine format is converted using a specification defined by a user.

10. The software testing method of any of the preceding claims further comprising:

compiling the first source code segment and the second source code segment into an object program; and
executing the object program to provide the event driven system.

11. The software testing method of any of the preceding claims wherein the step of testing uses at least one set of properties defined by a user associated with the execution of the event driven system.

12. The software testing method of claim 11 wherein the translation map is specified by the user.

13. The software testing method of claim 12 wherein the translation map, the environmental model, and the series of programming statements are each written in the C programming language.

14. The software testing method of any of the preceding claims including the step of displaying a result of the testing of the software.

15. The software testing method of claim 14 wherein the displayed result includes at least one full automaton repre-

sentation of the event driven system.

16. A software testing apparatus comprising means arranged to carry out each step of a method as claimed in any of the preceding claims.

17. A machine-readable medium having stored thereon a plurality of instructions, the plurality of instructions including instructions that, when executed by a machine, cause the machine to perform a method as claimed in any of claims 1 to 15.

18. A software testing system comprising:

a computer having at least one memory;
means for receiving a source program of software to be tested and comprising series of programming language statements in the memory;
and means arranged to carry out each step of a method as claimed in any of claims 1 to 15 to test said software.

FIG. 1

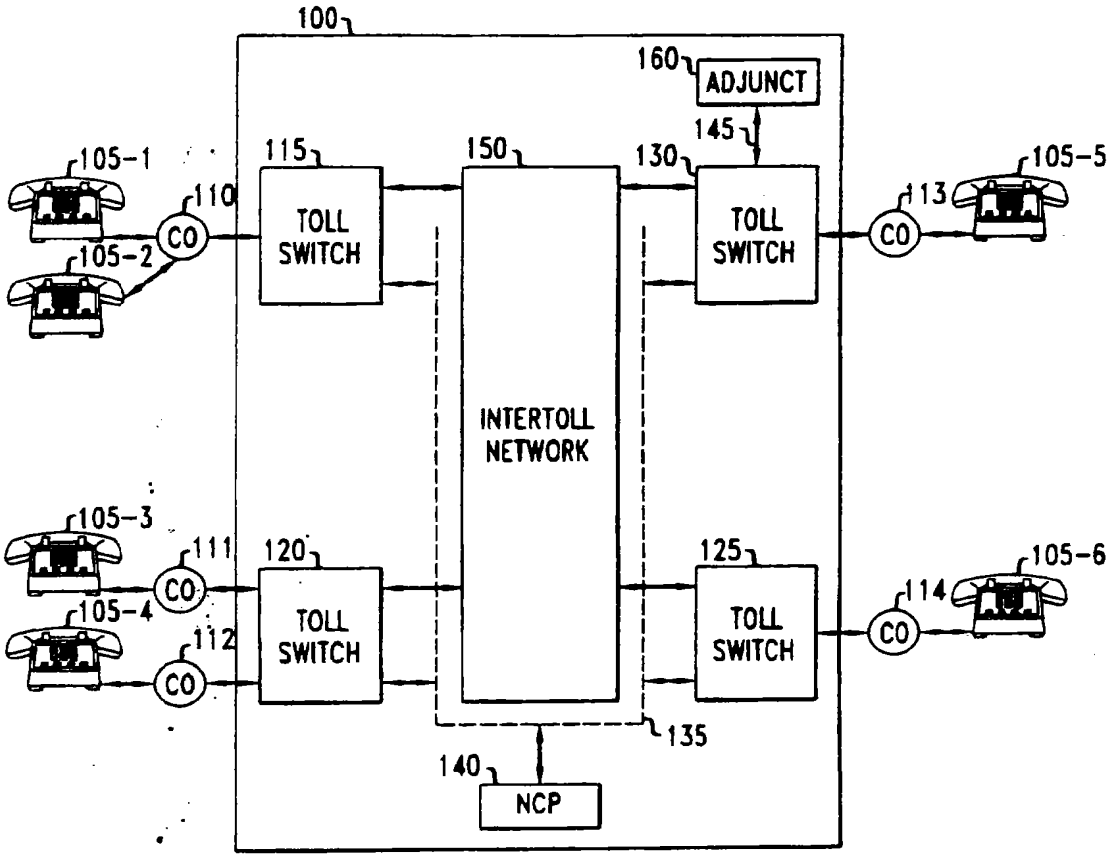

100
160
ADJUNCT
145
105-1
115
150
130
105-5
110
TOLL SWITCH
INTERTOLL NETWORK
TOLL SWITCH
113
CO
105-2
105-3
111
120
125
105-6
CO
TOLL SWITCH
TOLL SWITCH
114
105-4
112
CO
135
140
NCP

FIG. 2

```
1  //equivalent of state.z in traditional notation
2  //first switch on current state, then on event
.
.
.
69   default:
70     print ("no such state %d, line %d\n", x->lineno);
71   break;
72
73  case busy:   //phone is connected to a tone untill hung up
74   switch(op) {
75   default:
76     goto error;
77
78   case Crrtone:
79     x->state = busy
80     break;
81
82   case Cronhook:
83     send(x, Ctdis, 0);
84     x->state = idle;
85     break;
86   }
87   break;
.
.
.
```

205

```
.
.
.
89  case conn:          //phones are connected
90    switch(op) {
91    default:
92    case Crconn:      //successful connection
93      break;          // no state change
94
95    case Cronhook:
96      send(x->term, Ctdis, 0);
97      send(x, Ctdis, o);
98      x-.state = idle;
99      break;
100   }
101   break;

103 case dead;           //phone unassigned
104
105   switch(op) {      //check the event
106   default:
107       print("%l dead and op=%s\n", x, istring(op));
108   break;            //no state change
109
110   case Crprovlc:    //line provisioned
111   x->term = 0;      //initialize
112   x->state = idle;  //next state
113   break;
114   }
115   break;
```

210

215

TO FIG 3

200

FIG. 3

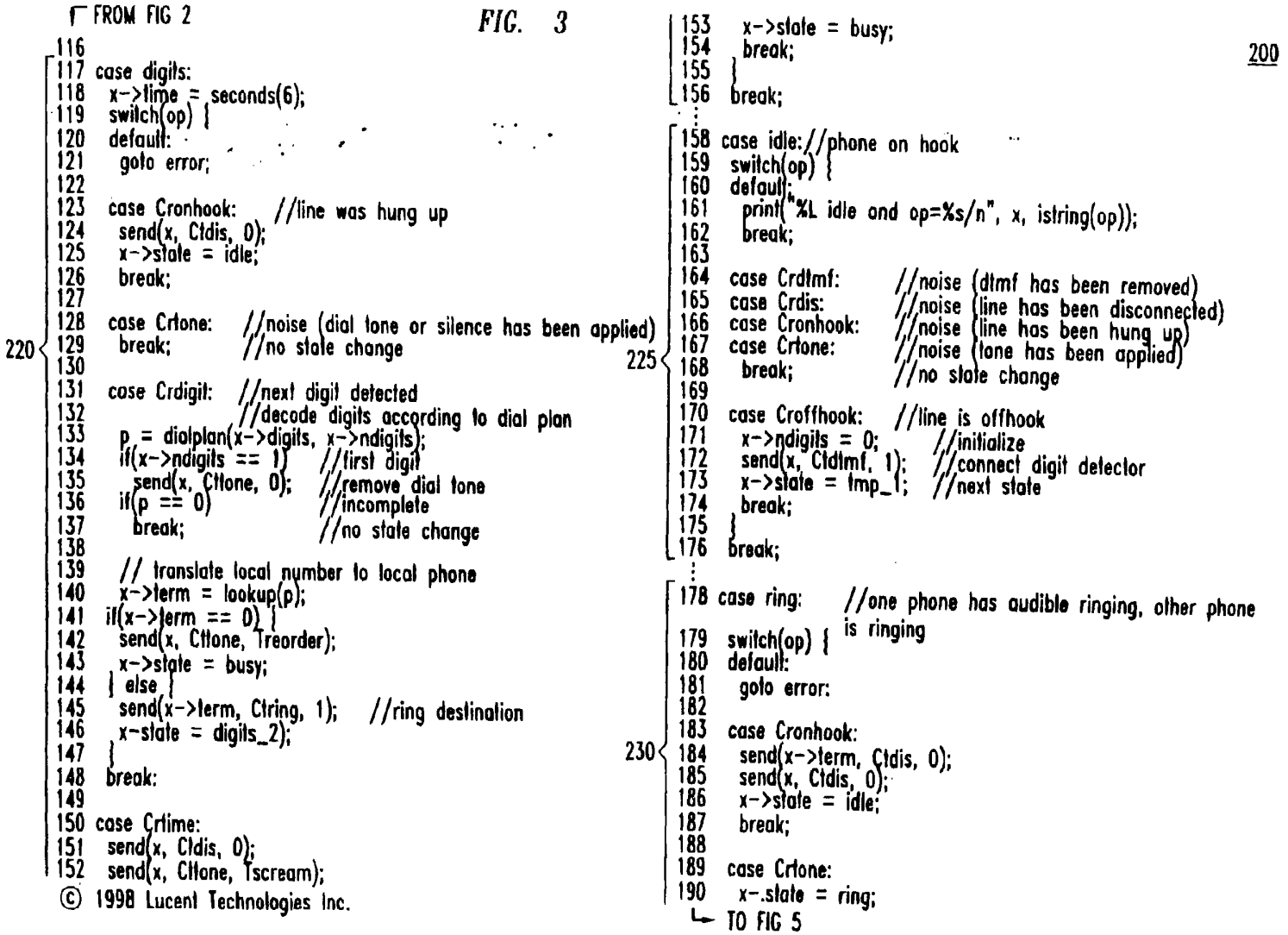

FROM FIG 2
116
117 case digits:
118 x->time = seconds(6);
119 switch(op) {
120 default:
121 goto error;
122
123 case Cronhook: //line was hung up
124 send(x, Ctdis, 0);
125 x->state = idle;
126 break;
127
128 case Crtone: //noise (dial tone or silence has been applied)
129 break; //no state change
130
131 case Crdigit: //next digit detected
132 //decode digits according to dial plan
133 p = dialplan(x->digits, x->ndigits);
134 if(x->ndigits == 1) //first digit
135 send(x, Cttone, 0); //remove dial tone
136 if(p == 0) //incomplete
137 break; //no state change
138
139 // translate local number to local phone
140 x->term = lookup(p);
141 if(x->term == 0) {
142 send(x, Cttone, Treorder);
143 x->state = busy;
144 } else {
145 send(x->term, Ctring, 1); //ring destination
146 x-state = digits_2);
147 }
148 break;
149
150 case Crtime:
151 send(x, Ctdis, 0);
152 send(x, Cttone, Tscream);
© 1998 Lucent Technologies Inc.
153 x->state = busy;
154 break;
155 }
156 break;
158 case idle://phone on hook
159 switch(op) {
160 default:
161 print("%L idle and op=%s/n", x, istring(op));
162 break;
163
164 case Crdtmf: //noise (dtmf has been removed)
165 case Crdis: //noise (line has been disconnected)
166 case Cronhook: //noise (line has been hung up)
167 case Crtone: //noise (tone has been applied)
168 break; //no state change
169
170 case Croffhook: //line is offhook
171 x->ndigits = 0; //initialize
172 send(x, Ctdtmf, 1); //connect digit detector
173 x->state = tmp_1; //next state
174 break;
175 }
176 break;
178 case ring: //one phone has audible ringing, other phone is ringing
179 switch(op) {
180 default:
181 goto error:
182
183 case Cronhook:
184 send(x->term, Ctdis, 0);
185 send(x, Ctdis, 0);
186 x->state = idle;
187 break;
188
189 case Crtone:
190 x-.state = ring;
TO FIG 5
200
220
225
230

FIG. 4

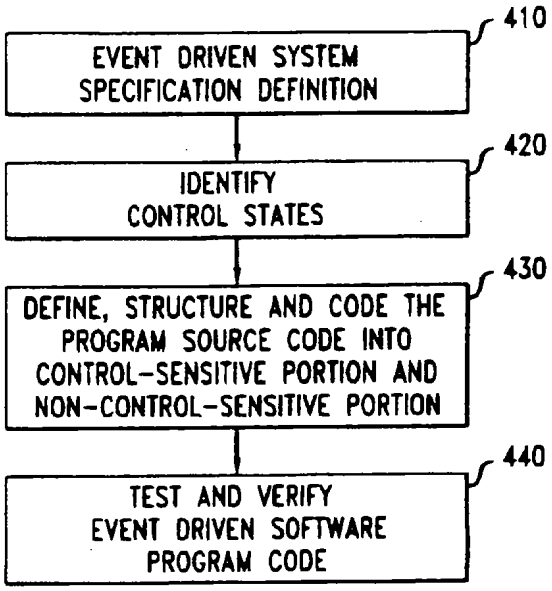

410
EVENT DRIVEN SYSTEM SPECIFICATION DEFINITION
420
IDENTIFY CONTROL STATES
430
DEFINE, STRUCTURE AND CODE THE PROGRAM SOURCE CODE INTO CONTROL-SENSITIVE PORTION AND NON-CONTROL-SENSITIVE PORTION
440
TEST AND VERIFY EVENT DRIVEN SOFTWARE PROGRAM CODE

FIG. 5

```
┌ FROM FIG 3
191
192
193  Cranswer:   //cross connect the phones
194    send(x, Ctconn, x->term->lineno);
195    send(x->term, Ctconn, x->lineno);
196    x->state = conn;
197    break;
198  }
199  break;
     ⋮
  1
  2
  3 #include  <u.h.>
     ⋮
 69  */
 70 @dead
 71  switch(op) {
 72  default:
 73    print("%L dead and op=%s\n", x, istring(op));
 74    goto Bdead;
 75
510 76  case Crprovlc:  //line is successfully provisioned
 77    break;
 78  }
 79 x->term = 0;
 80
 81 /*
     ⋮
 84 @idle:
 85  switch(op) {
 86  default:
 87    print("%L idle and op=%s\n", x, istring(op));
 88    goto Bidle;
 89
 90  case Crdtmf:     //noise (dtmf has been removed)
515 91  case Crdis:      //noise (line has been disconnected)
 92  case Cronhook:   //noise (line has been hung up)
 93  case Crtone:     //noise (tone has been applied)
 94    goto Bidle:    //noise (tone has been applied)
© 1998 Lucent Technologies Inc.
```

```
 95
 96  case Croffhook:                                    500
 97    break;
 98  }
 99 x->ndigits = 0;
100 send (x, Ctdtmf, 1);
     ⋮
102 @: switch(op) {
103    default:
104      goto error;
520 105
    106    case Cronhook:      //connect digit detector
    107      send(x, Ctdis, 0);
    108      goto Bidle;
    109
    110    case Crdtmf:        //digit detector is active
     ⋮
112  @: switch(op) {
113     default:
114       goto error;
115     case Cronhook:        //line was hung up
116       send(x, Crdis, 0);
117       goto Bidle;
530 118     case Crdigit:       //first digit (even before giving dial tone)
119       goto Adigits;
120     case Crtone:          //dial tone has been applied
121       break;
122   }
123 }
     ⋮
125 @digits
126  x->time = seconds(6);
127  switch(op) {
128  default:
535 129    goto error;
    130
    131  case Cronhook:        //line was hung up
    132    send(x, Ctdis, 0);
    133    goto Bidle;
    134
└ TO FIG 6
```

FIG. 6

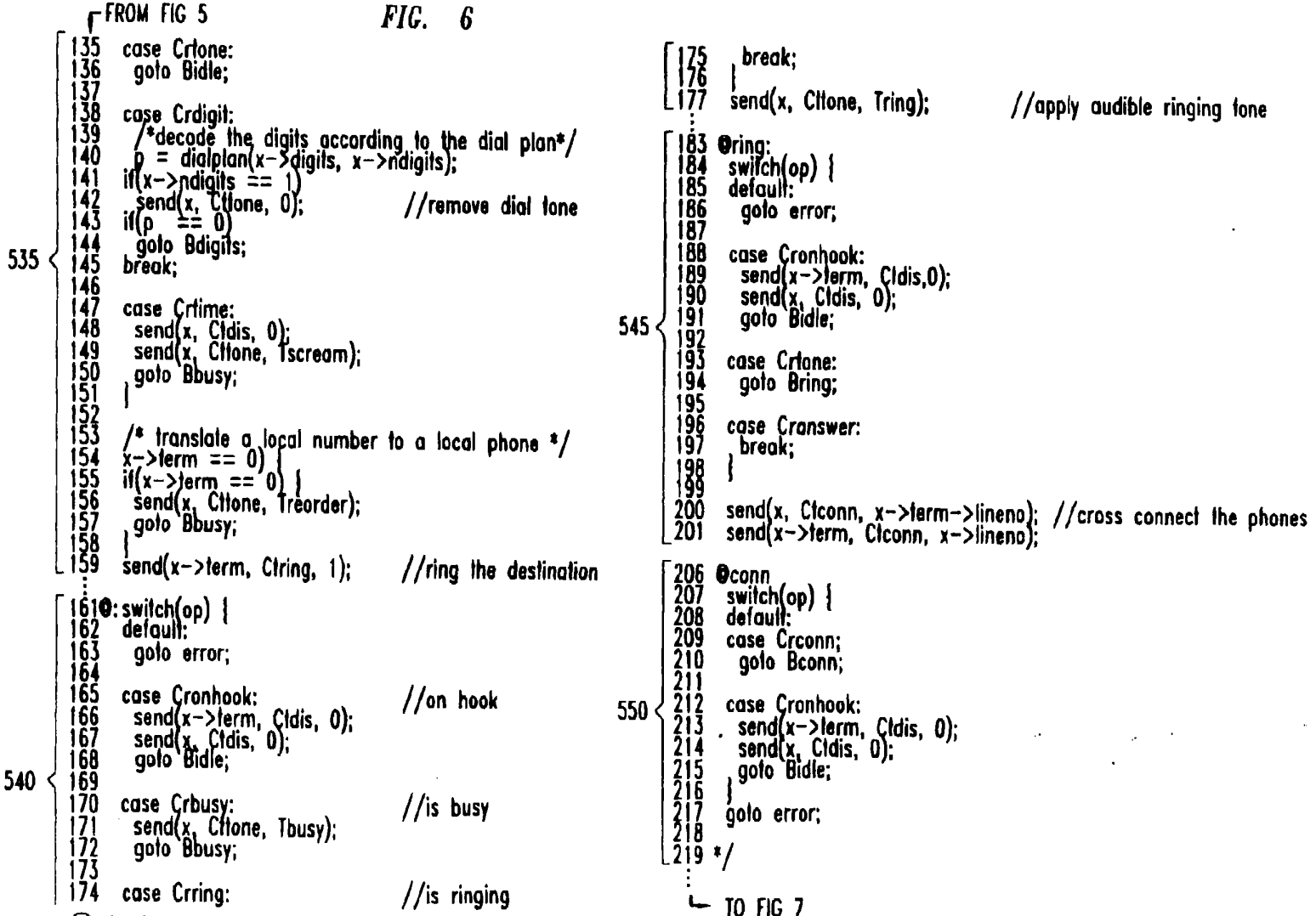


FIG. 7

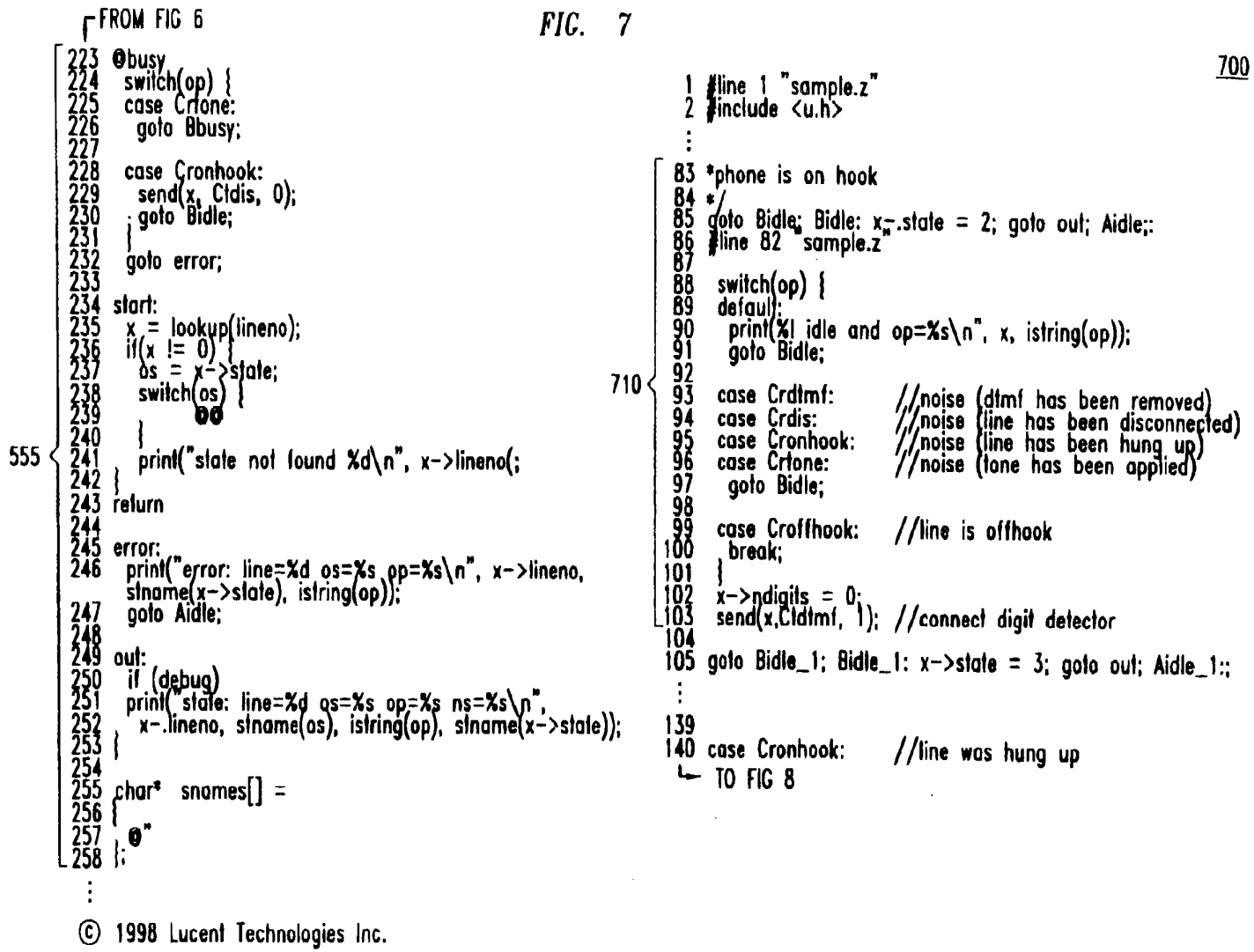


FIG. 8

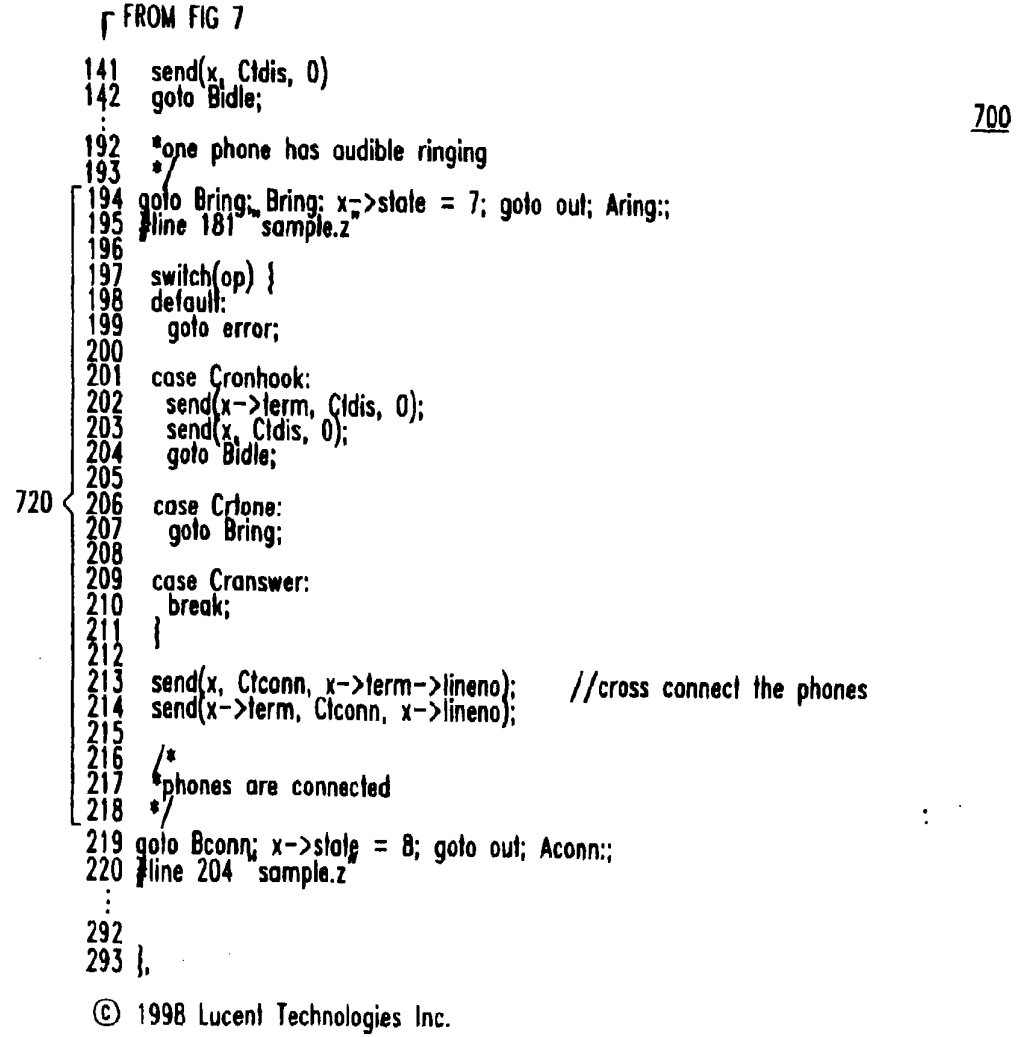


FIG. 9

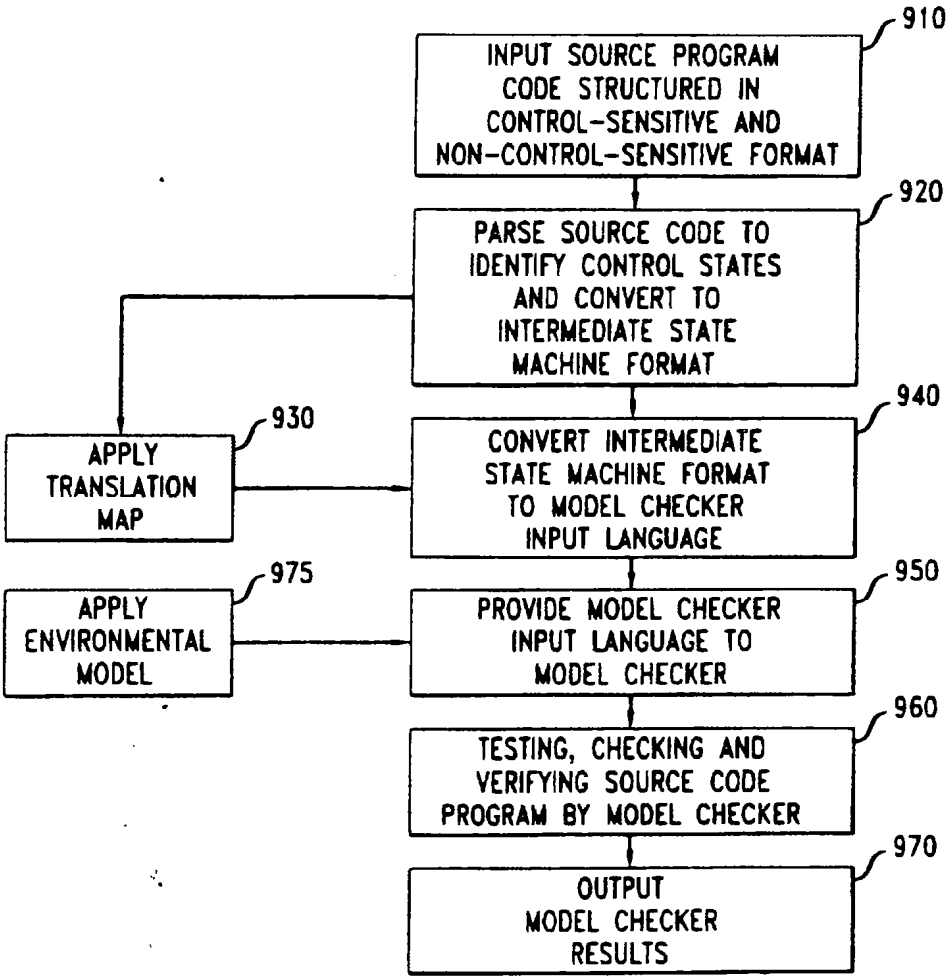

910
INPUT SOURCE PROGRAM CODE STRUCTURED IN CONTROL-SENSITIVE AND NON-CONTROL-SENSITIVE FORMAT
920
PARSE SOURCE CODE TO IDENTIFY CONTROL STATES AND CONVERT TO INTERMEDIATE STATE MACHINE FORMAT
930
APPLY TRANSLATION MAP
940
CONVERT INTERMEDIATE STATE MACHINE FORMAT TO MODEL CHECKER INPUT LANGUAGE
975
APPLY ENVIRONMENTAL MODEL
950
PROVIDE MODEL CHECKER INPUT LANGUAGE TO MODEL CHECKER
960
TESTING, CHECKING AND VERIFYING SOURCE CODE PROGRAM BY MODEL CHECKER
970
OUTPUT MODEL CHECKER RESULTS

FIG. 10

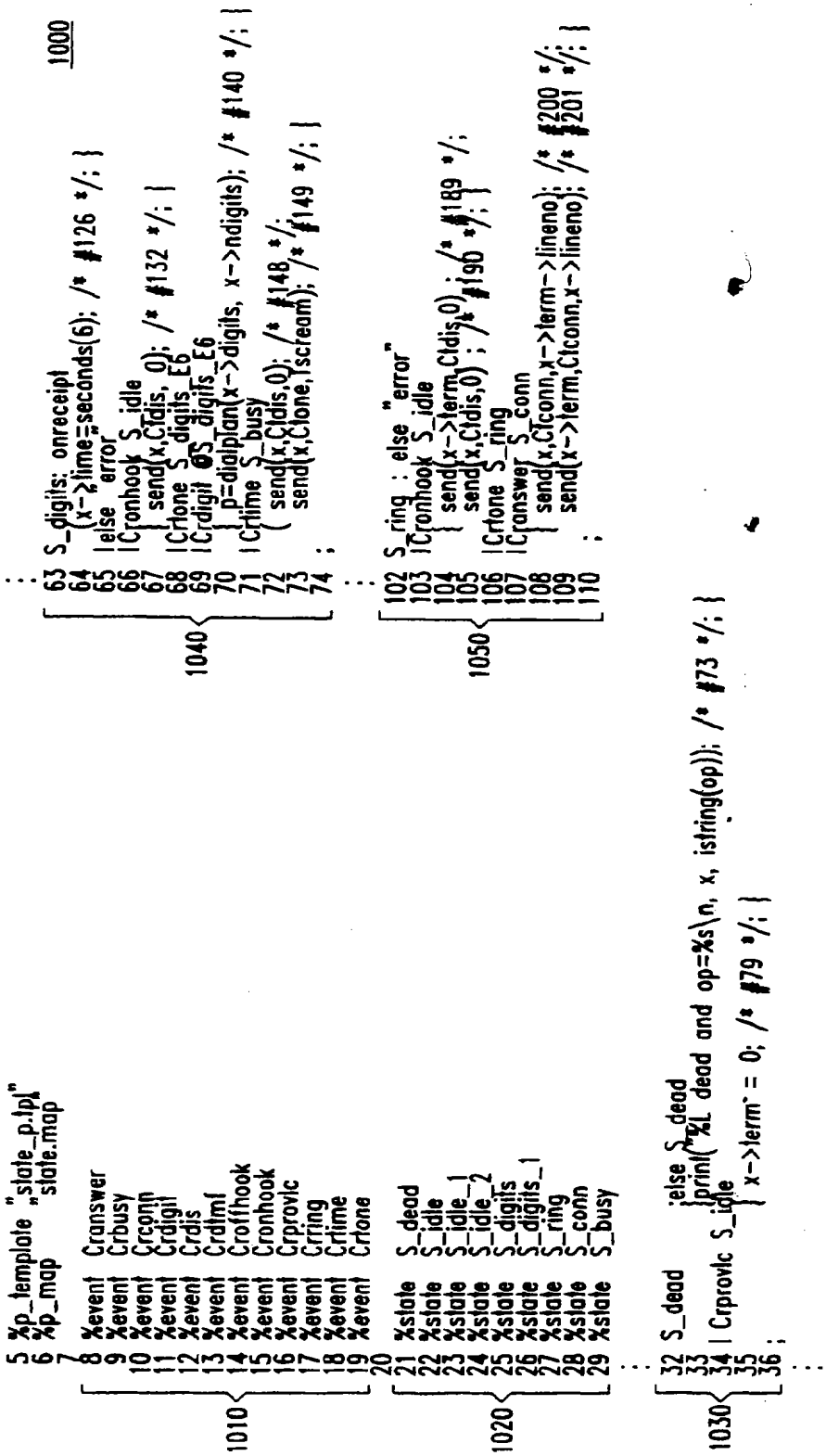


FIG. 11

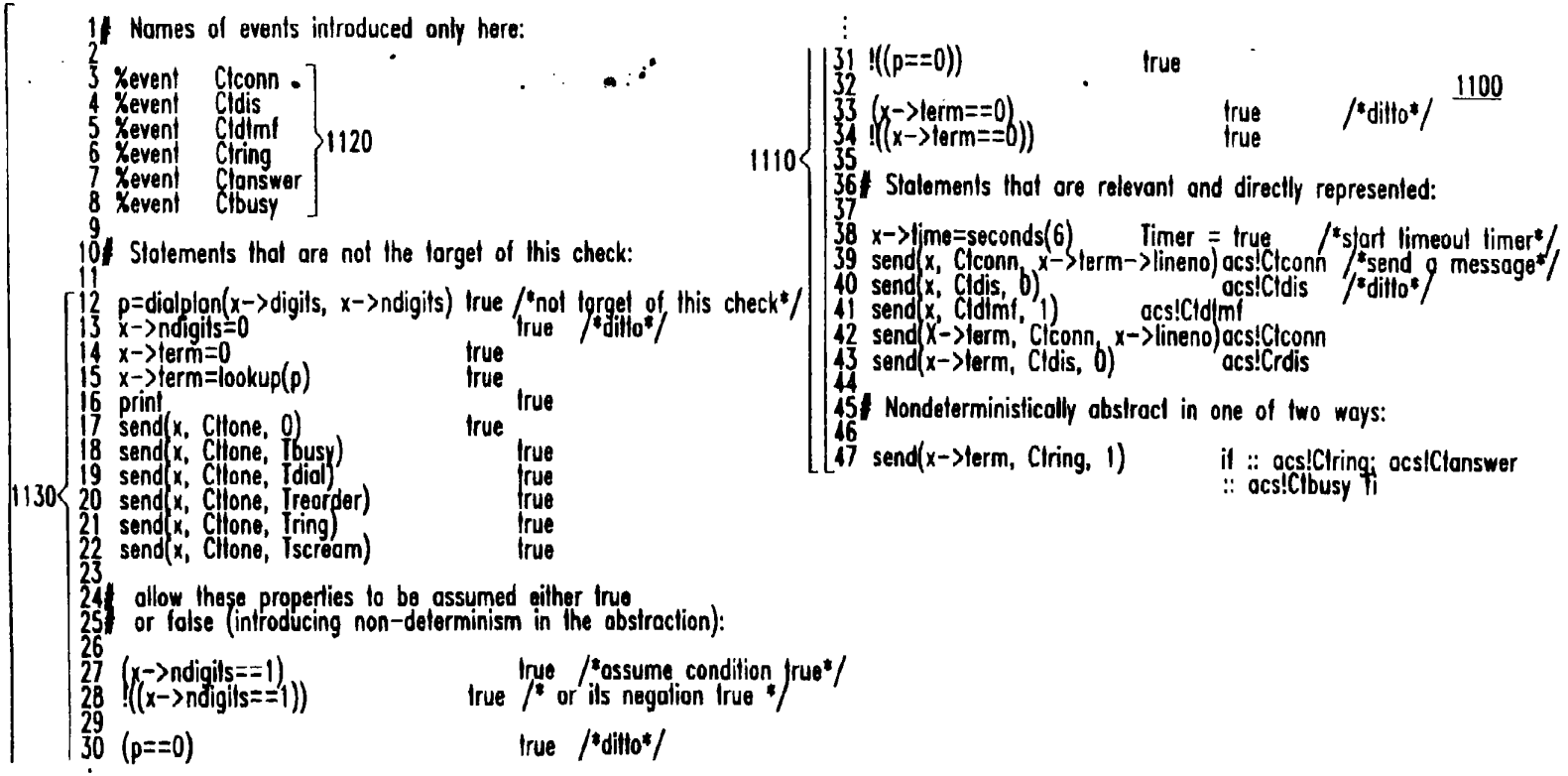


FIG. 12

1200

```
 1  /*
 2   * source: state */
 .
 .
 .
139  A_S_digits_Cronhook;
140     /* { send(x, Ctdis, 0); ** #132 **; } */
 |
TO FIG 13
```

FIG. 13

```
FROM FIG 12                                          1200
    |
   141    goto BS_idle;
   142  :: else ->
    .
    .
    .
   302  qin'Crprovlc          /*provision the phone*/
   303  }
```

FIG. 14

1400

```
1   /*
2   *source: state.x
.
.
.
152 od
153 }
```

FIG. 15

1500

```
1   /* Spin template */
2
:
19  prototype context ()
20  {   mtype msg;
21
22  end:    do
23    :: Timer && tineout -> /* the timer expires */
24          Timer = false;   /* turn it off */
25          qin!Crtime       /* notify the switch */
26
27    :: atomic {
28          acs?msg->        /* message sent to enviornment */
29          Timer = false;   /* on arrival of any event: timer cancels */
30
31          if
32          :: msg == Ctdtmf ->     /* digit recognition */
33                qin!Crdtmf;       /* must acknowledge this event */
34                dialtone = 1 - dialtone /* toggle on-off
35
36          :: msg == Ctdis ->
37                qin!Crdis;        /* acknowledge */
38                dialtone = false  /* disconnecting - dialtone off */
39
40          :: msg ==Ctring ->      /* phone rings */
41                qin!Crring
42
43          :: msg == Ctanswer ->   /* phone answered */
44                qin!Cranswer
45
46          :: msg == Ctbusy ->     /* phone is busy */
47                qin!Crbusy
48
49          :: msg == Crdis    msg == Ctconn    msg == Crconn ->
50                skip   /* messages not yet considered */
51
52          :: else ->              /* nothing else should happen */
53                printf(*context: cannot happen\n");
54                assert(false)  /* assert that we cannot reach this */
55          fi
56        }
57    od
58  }
:
70    od
```

1510

FIG. 16

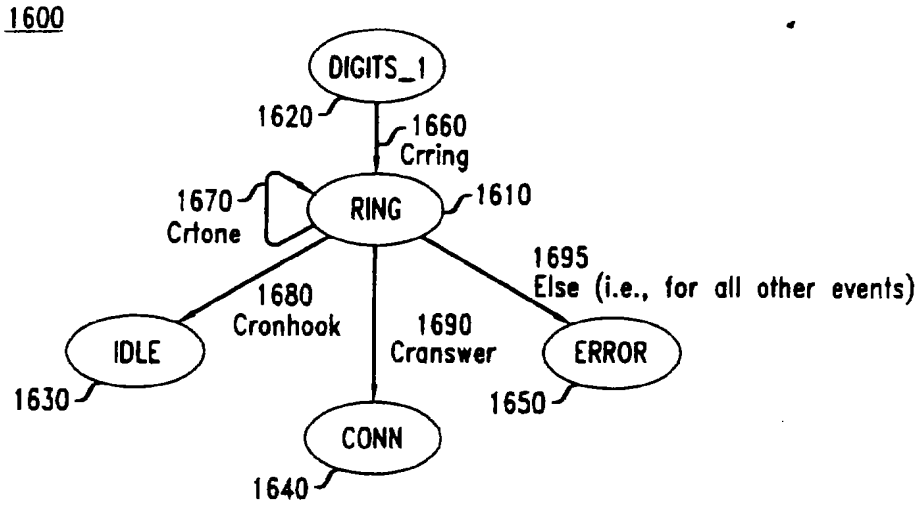

1600
DIGITS_1
1620
1660
Crring
1670
Crtone
RING
1610
1695
Else (i.e., for all other events)
1680
Cronhook
1690
Cranswer
IDLE
1630
ERROR
1650
CONN
1640

FIG. 17

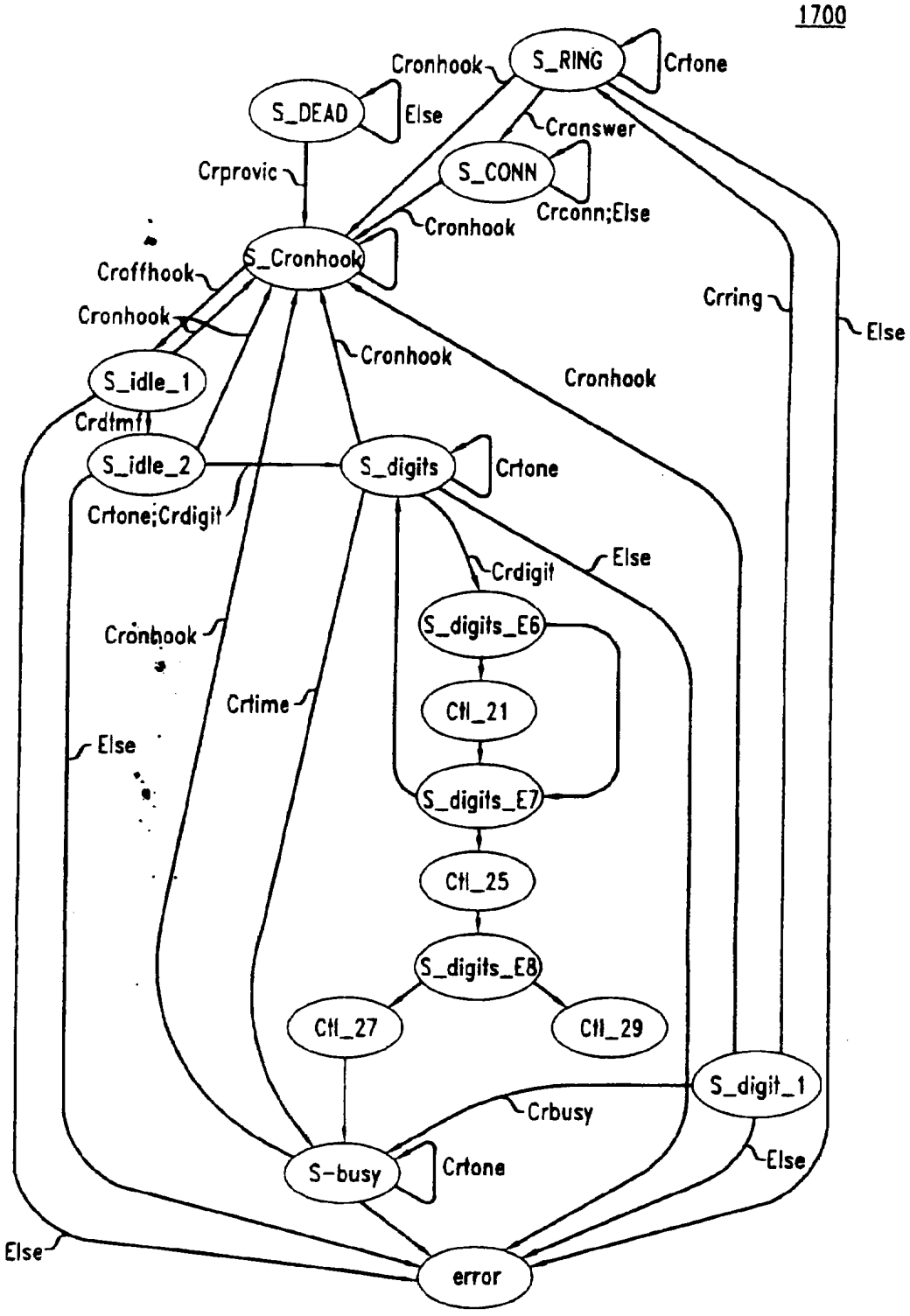

1700
S_RING
Crtone
Cronhook
S_DEAD
Else
Cranswer
S_CONN
Crconn;Else
Crprovic
Cronhook
S_Cronhook
Croffhook
Cronhook
Crring
Else
Cronhook
Cronhook
S_idle_1
Crdtmf
S_idle_2
S_digits
Crtone
Crtone;Crdigit
Else
Crdigit
S_digits_E6
Cronhook
Crtime
CH_21
Else
S_digits_E7
CH_25
S_digits_E8
CH_27
CH_29
S_digit_1
Crbusy
S-busy
Crtone
Else
Else
error

FIG. 18

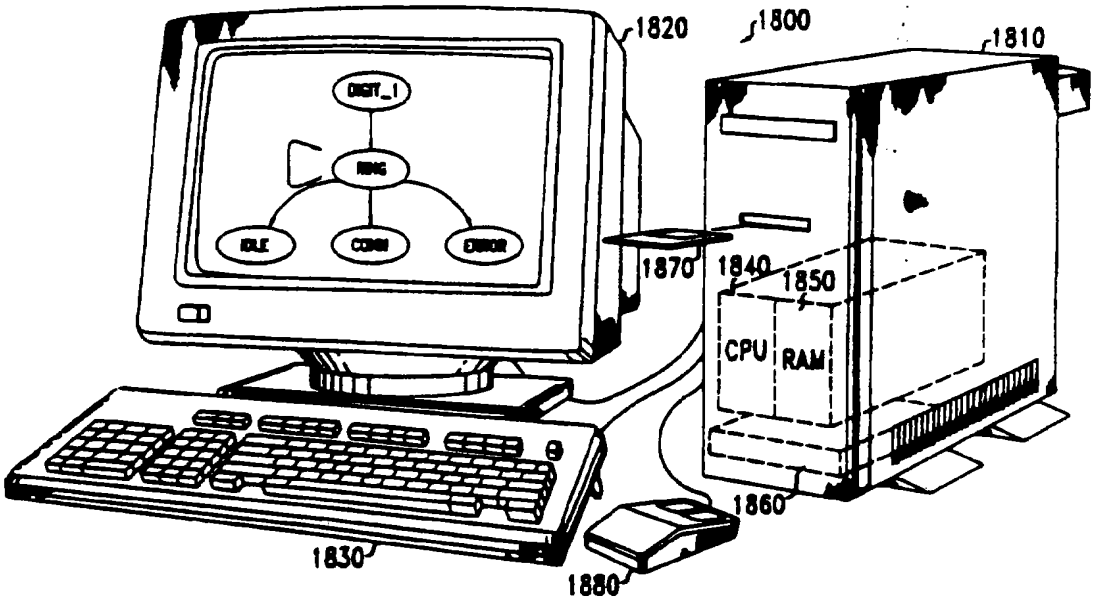
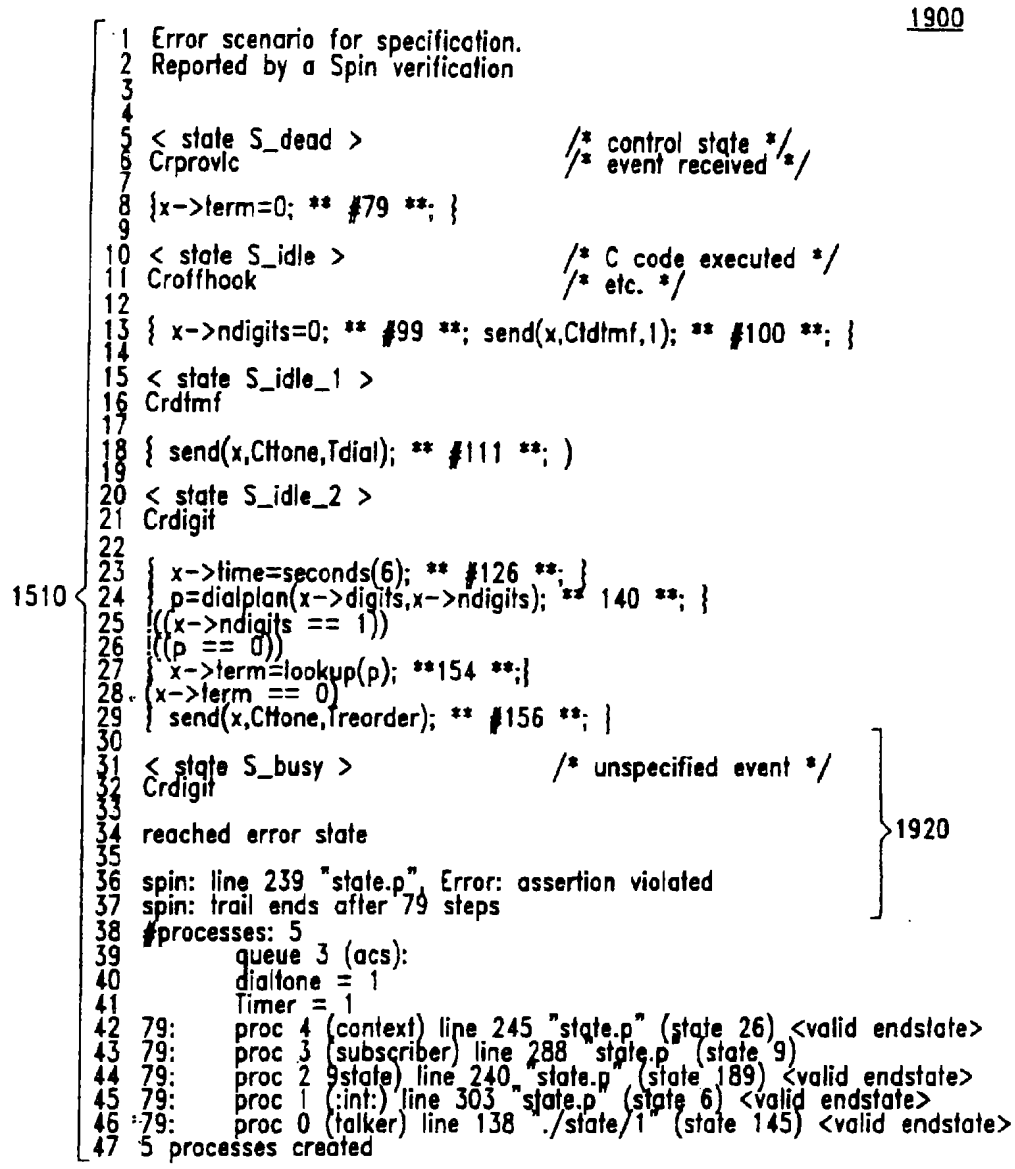


FIG. 19



European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 99 30 9818

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
X	EP 0 685 792 A (AT&T CORP.) 6 December 1995 (1995-12-06) * column 1, line 35 - line 56 *	1-18	G06F11/00
A	GERARD J. HOLZMANN: "The Model Checker SPIN" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING., vol. 23, no. 5, May 1997 (1997-05), pages 279-295, XP000720371 IEEE INC. NEW YORK., US ISSN: 0098-5589 * page 280, left-hand column, line 21 - right-hand column, line 19 *	1-18	

TECHNICAL FIELDS SEARCHED (Int.Cl.7)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	3 May 2000	Corremans, G

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 99 30 9818

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

03-05-2000

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 685792 A	06-12-1995	CA 2147536 A	02-12-1995
		JP 7334566 A	22-12-1995
		US 5615137 A	25-03-1997

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82